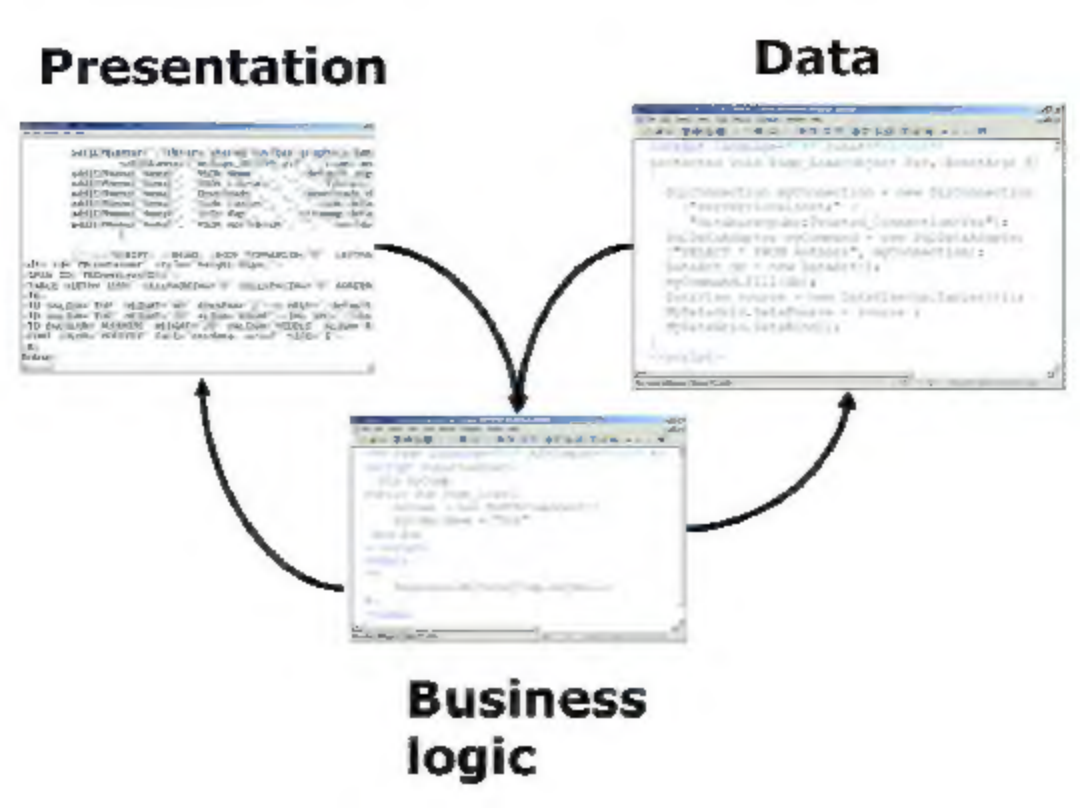
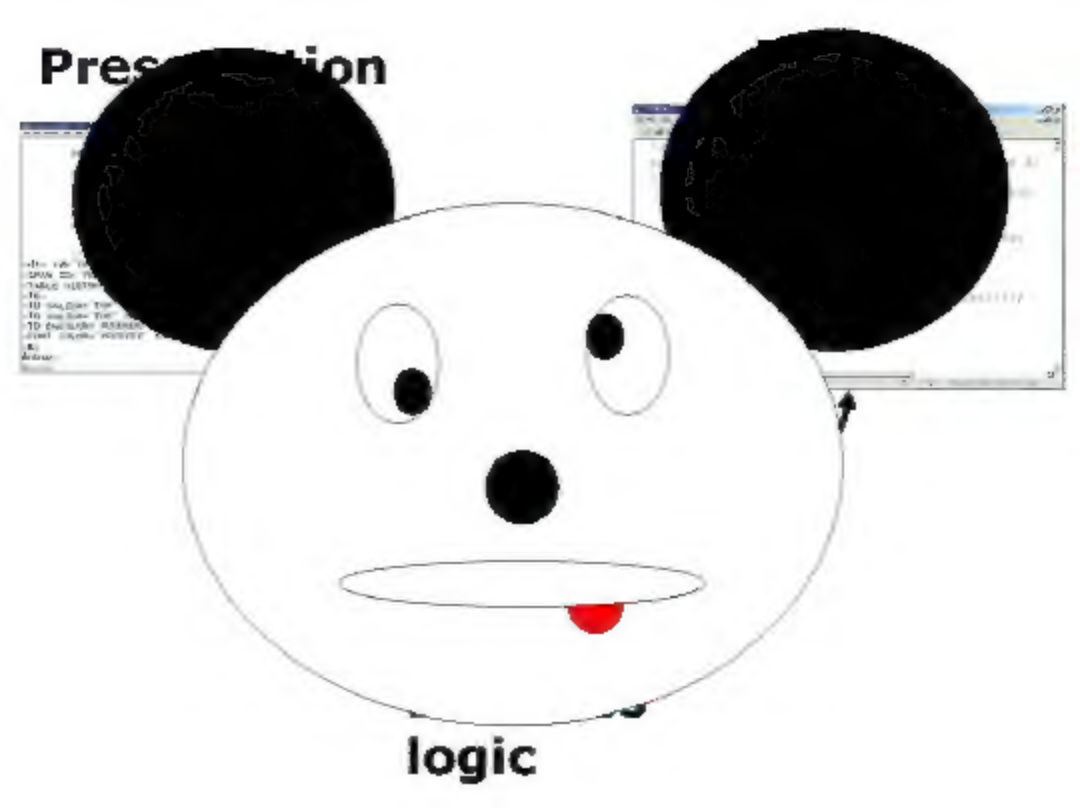
From Haskell via Cω to LINQ

A Personal Perspective

Erik Meijer


Presentation
Data
Business logic


Pres tion
logic


Presentation
(*HaskellScript*)
Business
logic
(*HSP,*
XMλ)
Data
(*Haskell*
DB)

The Haskell Days

Swartz, SETL 1970; Burstall and Darlington, NPL 1977; Turner, KRC 1981; .... Haskell, Python, Scala, ...

Write programs using mathematical ZF set comprehensions syntax.

Wadler 1989

List comprehensions are related to the relational calculus.

Wadler 1992

List comprehensions are a special case of monad comprehensions.

List Comprehensions Example

```
factors n =
  [ x                  Select
  | x <- [1..n]        From
  , n `mod` x == 0     Where
  ]

isPrime n = factors n = [1,n]

primes n =
  [ p | p <- [2..n], isPrime p ]
```

The Haskell Days

Swartz, SETL 1970; Burstall and Darlington, NPL 1977; Turner, KRC 1981; .... Haskell, Python, Scala, ...

Write programs using mathematical ZF set comprehensions syntax.

Wadler 1989

List comprehensions are related to the relational calculus.

Wadler 1992

List comprehensions are a special case of monad comprehensions.

List Comprehensions Example

```
factors n =
  [ x                  Select
  | x <- [1..n]        From
  , n `mod` x == 0     Where
  ]

isPrime n = factors n = [1,n]

primes n =
  [ p | p <- [2..n], isPrime p ]
```

List Comprehensions
Simplified translation

```
[ e | True ] =
  [ e ]
[ e | q ] =
  [ e | q, True ]
[ e | b, Q ] =
  filter b [ e | Q ]
[ e | x <- l, Q ] =
  concatMap (\x -> [e | Q ]) l
[ e | let decls, Q ] =
  let decls in [ e | Q ]
```

Syntactic sugar over standard list operations

Monad Comprehensions Example

```
words :: IO [String]
words =
  do{ putStr "enter a value …"
    ; x <- getLine
    ; return (words x)
    }

class Monad m where
  { (>>=) :: m a -> (a -> m b) -> m b
  ; return :: a -> m a
  }
```

Parametrized over type constructor

Syntactic sugar over standard monad operations

List Comprehensions
Simplified translation

```
[ e | True ] =
  [ e ]
[ e | q ] =
  [ e | q, True ]
[ e | b, Q ] =
  filter b [ e | Q ]
[ e | x <- l, Q ] =
  concatMap (\x -> [e | Q ]) l
[ e | let decls, Q ] =
  let decls in [ e | Q ]
```

Syntactic sugar over standard list operations

Monad Comprehensions Example

```
words :: IO [String]
words =
  do{ putstr "enter a value  :"
    ; x <  getLine
    ; return (words x)
    }

class Monad m where
  { (>>=) :: m a -> (a -> m b) -> m b
  ; return :: a -> m a
  }
```

Syntactic sugar over standard monad operation

HaskellDb Query Monad

```
SELECT X.FirstName, X.LastName
FROM Authors AS X
WHERE X.City = 'OakLand'
```

```
oaklands =
  do{ x <- table authors
    ; restrict (x!city .==. constant "Oakland")
    ; project ( au_fname = x!au_fname
              , au_lname = x!au_lname
              )
    }
```

Query monad

Monad for expressions

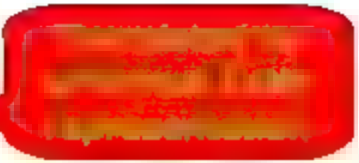
Haskell Server Pages
XHTML Literals

```
table :: TABLE
table = <TABLE border="1 >
          <% mkRows cells %>
        </TABLE>

cells :: [[(Int,Int)]]
cells = [[ (x,y) | x <- [1..16] ] | y <- [1..16] ]

mkRows :: [[(Int,Int)]] -> [TR]
mkRows =
  map $ \cs -> <TR><% mkColums cs %></TR>

mkColumns :: [(Int,Int)] -> [TD]
mkColums =
  map $ \c -> <TD bgcolor=(color c)><% c %></TD>
```



ASP.net, embedding

Growing Cω

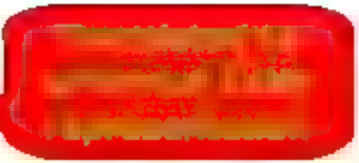
Haskell Server Pages
XHTML Literals

```
table :: TABLE
table = <TABLE border="1 >
          <% mkRows cells %>
        </TABLE>

cells :: [[(Int,Int)]]
cells = [[ (x,y)   x <- [1..16] ] | y <- [1..16] ]

mkRows :: [[(Int,Int)]] -> [TR]
mkRows =
  map $ \cs -> <TR><% mkColums cs %></TR>

mkColumns :: [(Int,Int)] -> [TD]
mkColums =
  map $ \c -> <TD bgcolor=(color c)><% c %></TD>
```



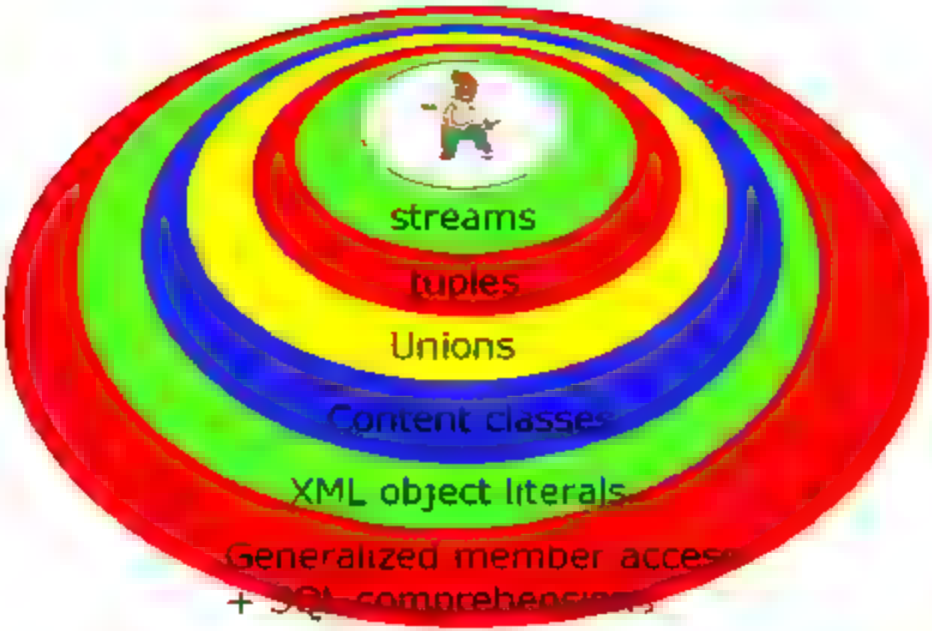
Growing Cω

Growing Cω

Growing Cω

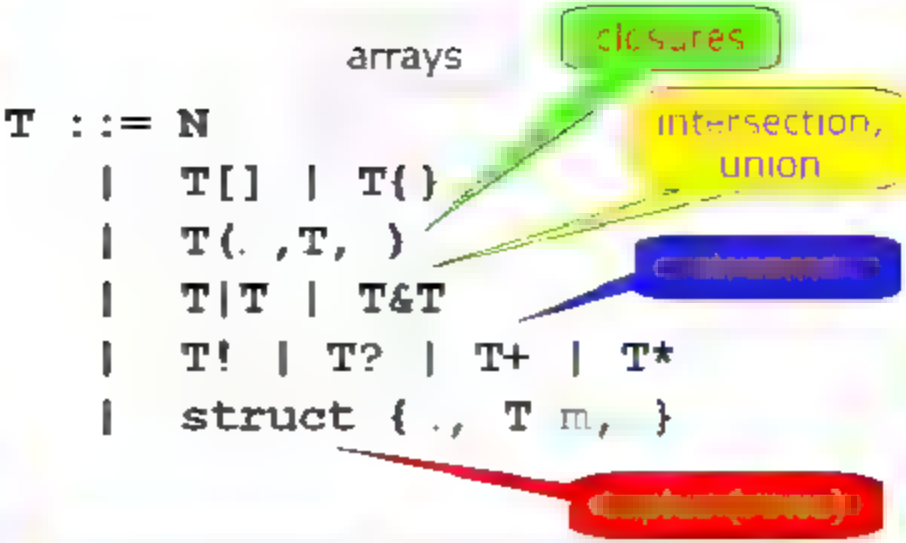
Growing Cω

Growing Cω


streams
tuples
Unions
Content classes
XML object literals
Generalized member acces
+ SQL comprehensions

Type System Extensions

(structural)



Cω
Query Comprehensions

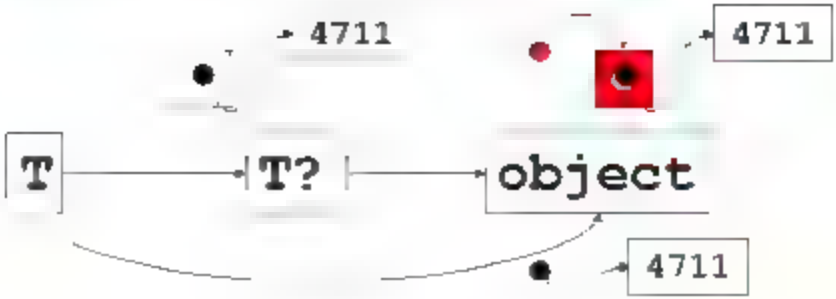
```
String n = "Wolfram";
struct{String? Subject}* subjects =
  select it.Subject from it in inbox
  where it.From == n

foreach(r in
  select CustomerID, ContactName
  from dbo.Customers where City == mycity
  order by ContactName) {
  …
}
```

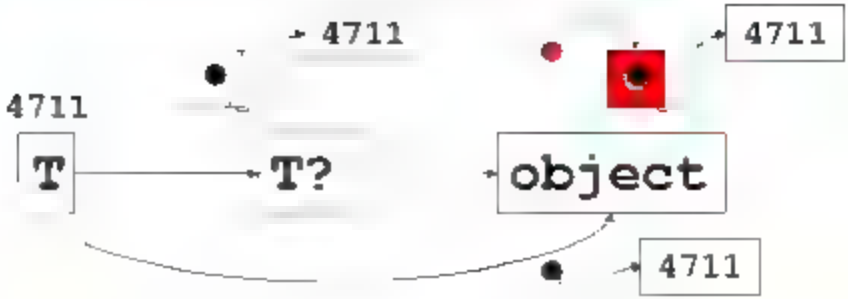
Compiler plugins

Type inference

Incoherence?



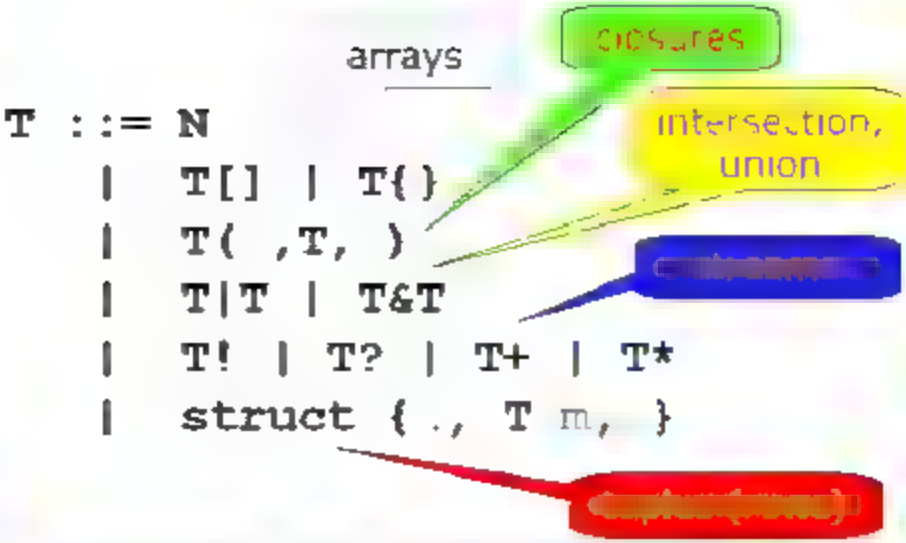
Incoherence?



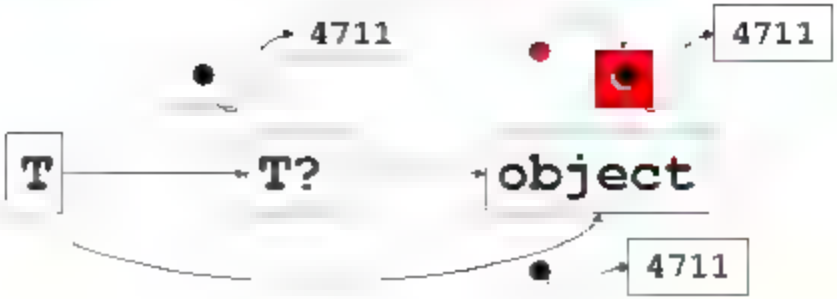
When up-casting from T? to object we need to unwrap (➔ in Whidbey)

Type System Extensions

(structural)



Incoherence?



LINQ Project

Query Comprensions In C# 3.0

```
var contacts =
  from c in customers
  where c.State == "WA"
  select new { c.Name, c.Phone };
```

Type inference

Syntactic sugar over standard query operations

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c => new{c.Name, c.Phone});
```

Lambda expression

Extension Methods

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c -> new{c.Name, c.Phone});

Static class System.Query
{
    public static IEnumerable<T>
      Where<T>(this IEnumerable<T> src,
               Func<T, bool>> p);
}
```

IEnumerable<Customer>

Extension method for IEnumerable<T>

Expression Trees

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c => new{c.Name, c.Phone});

class Table<T>: IEnumerable<T>
{
    public Table<T>
      Where(Expression
            <Func<T, bool>> p);
    .
}
```

Table<Customer>

Intermediate representation of delegate

Extension Methods

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c -> new{c.Name, c.Phone});

Static class System.Query
{
    public static IEnumerable<T>
      Where<T>(this IEnumerable<T> src,
               Func<T, bool>> p);
}
```

IEnumerable<Customer>

Extension method for IEnumerable<T>

Expression Trees

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c => new{c.Name, c.Phone});

class Table<T>: IEnumerable<T>
{
    public Table<T>
      Where(Expression
            <Func<T, bool>> p);
    .
}
```

Table<Customer>

Intermediate representation of delegate

Cω
Query Comprehensions

```
String n = "Wolfram";
struct{String? Subject}* subjects =
  select it.Subject from it in inbox
  where it.From == n

foreach(r in
  select CustomerID, ContactName
  from dbo.Customers where City == mycity
  order by ContactName) {
  …
}
```

Compiler plugins

Type inference

HaskellDb Query Monad

```
SELECT X.FirstName, X.LastName
FROM Authors AS X
WHERE X.City = 'OakLand'
```

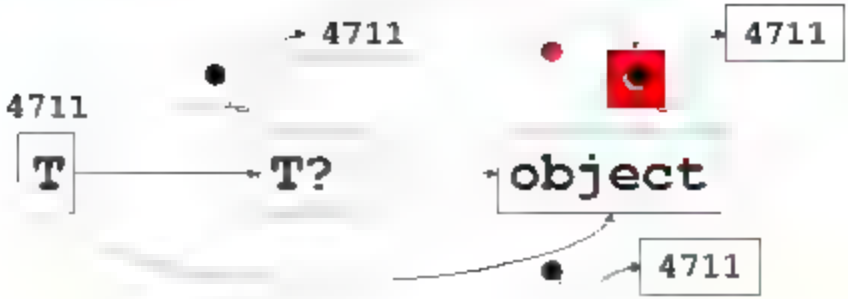
```
oaklands =
  do{ x <- table authors
    ; restrict (x!city .==. constant "Oakland")
    ; project ( au_fname = x!au_fname
              , au_lname = x!au_lname
              )
    }
```

Query monad

Intent

for expressions

Incoherence?



Expression Trees

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c => new{c.Name, c.Phone});

class Table<T>: IEnumerable<T>
{
    public Table<T>
      Where(Expression
            <Func<T, bool>> p);
    .
}
```

Table<Customer>

Internal representation of delegate

Anonymous types, Object initializers

```
var contacts =
  from c in customers
  where c.State == "WA"
  select new { c.Name, c.Phone };
```

Anonymous types

Object Initializers

```
var Joe = new Person{
   Name = "Joe", Age = 42,
   Address = { Street = "1th st",
               City = "Seattle" }
}
```

Expression Trees

```
var contacts =
  customers
  .Where(c => c.State == "WA")
  .Select(c => new{c.Name, c.Phone});

class Table<T>: IEnumerable<T>
{
    public Table<T>
      Where(Expression
            <Func<T, bool>> p);
    .
}
```

Table<Customer>

Internal representation of delegate

Anonymous types, Object initializers

```
var contacts =
  from c in customers
  where c.State == "WA"
  select new { c.Name, c.Phone };
```

Anonymous types

Object Initializers

```
var Joe = new Person{
   Name = "Joe", Age = 42,
   Address = { Street = "1th St",
               City = "Seattle" }
}
```

Query Comprensions In Visual Basic 9

SQL-style Select-From-Where
vs traditional From-Where-Select

```
Dim contacts =
  Select c.Name, c.Phone
  From c In customers
  Where c.State = "WA"
```

Parity with other C# 3.0 features

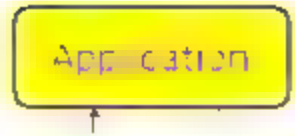
```
r   c    db.Customers
 .W.e(c => c.City == "London")
.Select
new { c.Name, c.Phone }
```

LINQ Query

SQL Query

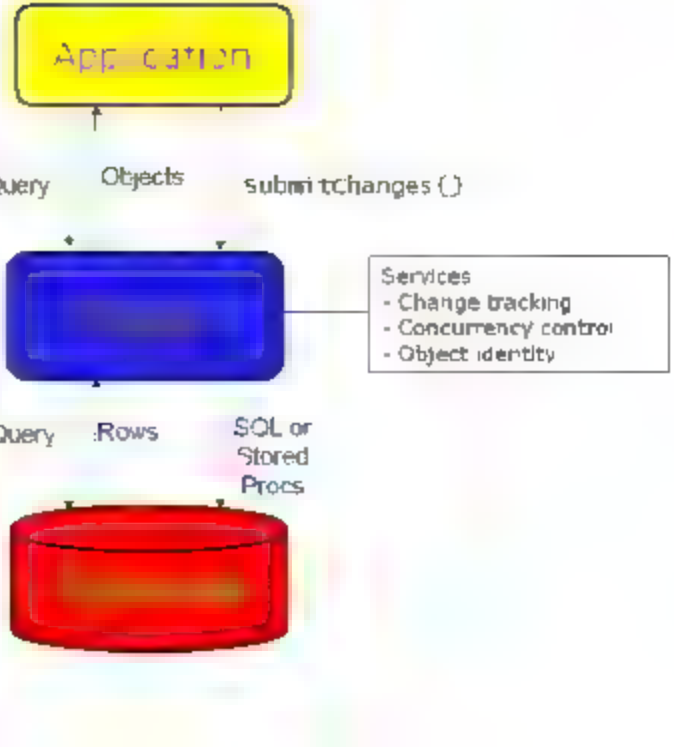
```
select Name, Phone
from Customers
where City = 'London'
```

```
c    db  ustomers
e (  it) ==  c
e
 new { c Name  c Phone }
```




```
 1    Name  Phone
 m Customers
 e e cit, =  London
```

```
db.Customers
(c.City == "London")
select
new { c.Name, c.Phone }
```




```
SELECT Name, Phone
FROM Customers
WHERE City = 'London'
```

DLinq
O/R Mapping

Mapping through attributes
Manually authored or tool generated
DataContext allows access to relational data as objects

Automatic change tracking
Auto-generated updates using optimistic concurrency

Works with existing infrastructure (ADO.Net)
Integrates with System.Transactions, SQL pass-through, returning objects from SQL queries.

XML DOM

```
Dim PO As New XmlDocument

Dim purchaseOrder As XmlElement =
    PO.CreateElement( "purchaseOrder" )
PO.AppendChild(purchaseOrder)

Dim orderDate As XmlAttribute =
  PO.CreateAttribute( "orderDate" )
orderDate.Value = "10/20/2001"
purchaseOrder.Attributes.Append(orderDate)

Dim shipTo As XmlElement =
PO.CreateElement( "shipTo" )
purchaseOrder.AppendChild(shipTo)

Dim country As XmlAttribute =
PO.CreateAttribute( "country" )
country.Value = "US"
shipTo.Attributes.Append(country)
```

What does this program do

XLinq API

Functional expression-based construction

```
Dim Item = _
New XElement("item", _
  New XAttribute("partNum", "926-AA"), _
  New XElement("productName", "..."), _
    New XElement("quantity", 1), _
    New XElement("price", 39.98), _
    New XElement("shipDate", "1999-05-21"))))
```

Simple

Context-free (no document scope)

XML DOM

```
Dim PO As New XmlDocument

Dim purchaseOrder As XmlElement =
    PO.CreateElement( purchaseOrder )
PO.AppendChild(purchaseOrder)

Dim orderDate As XmlAttribute =
  PO.CreateAttribute( orderDate )
orderDate.Value = [illegible]
purchaseOrder.Attributes.Append(orderDate)

Dim shipTo As XmlElement =
PO.CreateElement( shipTo )
purchaseOrder.AppendChild(shipTo)

Dim country As XmlAttribute =
PO.CreateAttribute( country )
country.Value = [illegible]
shipTo.Attributes.Append(country)
```

What does this program do?

XLinq API

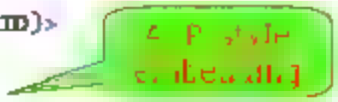
Functional 'expression-based' construction

```
Dim Item    _
New XElement( item", _
  New XAttribute('partNum", "926 AA"), _
  New XElement("productName", "..."), _
    New XElement( quantity", 1), _
    New XElement('price", 39.98), _
    New XElement('shipDate", "1999-05-21"))))
```

Simple

Context-free (no document scope)

```
Dim PO = <purchaseOrder orderDate=(System.DateTime.Today)>
           <shipTo country="US">
             <name>Alice Smith</name>
             <street>123 Maple Street</street>
             <city>Mill Valley</city>
             <state>CA</state>
             <zip>90952</zip>
           </shipTo>
           <%= billTo %>
           <items>
           <%=
             Select <item partNum=(O.PartID)>
                      <productName>
                        <%= O.Product %>
                      </productName>
                      <quantity>
                        <%= O.Quantity %>
                      </quantity>
                      <price>
                        <%= O.Price %>
                      </price>
                    </item>
             From O In Orders
             Where O.Name =  Robert Smith
           %>
           </items>
         </purchaseOrder>
```



XLinq API

Functional 'expression-based' construction

```
Dim Item = _
New XElement( item", _
  New XAttribute( partNum", "926 AA"), _
  New XElement("productName", "..."), _
    New XElement( quantity", 1), _
    New XElement('price", 39.98), _
    New XElement('shipDate", "1999-05-21"))))
```

Simple

Content-free (no document scope)

XML DOM

```
Dim PO As New XmlDocument

Dim purchaseOrder As XmlElement =
    PO.CreateElement( purchaseOrder )
PO.AppendChild(purchaseOrder)

Dim orderDate As XmlAttribute =
  PO.CreateAttribute( orderDate )
orderDate.Value = [illegible]
purchaseOrder.Attributes.Append(orderDate)

Dim shipTo As XmlElement =
PO.CreateElement( shipTo )
purchaseOrder.AppendChild(shipTo)

Dim country As XmlAttribute =
PO.CreateAttribute( country )
country.Value = [illegible]
shipTo.Attributes.Append(country)
```

What does this program do

```
Dim PO = <purchaseOrder orderDate=(System.DateTime.Today)>
           <shipTo country="US">
             <name>Alice Smith</name>
             <street>123 Maple Street</street>
             <city>Mill Valley</city>
             <state>CA</state>
             <zip>90952</zip>
           </shipTo>
           <%= billTo %>
           <items>
           <%=
             Select <item partNum=(O.PartID)>
                      <productName>
                        <%= O.Product %>
                      </productName>
                      <quantity>
                        <%= O.Quantity %>
                      </quantity>
                      <price>
                        <%= O.Price %>
                      </price>
                    </item>
             From O In Orders
             Where O.Name = "Robert Smith
           %>
           </items>
         </purchaseOrder>
```

Late binding over XML

Child axis

```
BillTo.street
```

➔ `BillTo.Elements("street")`

Attribute axis

```
BillTo.@country
```

➔ `BillTo.Attribut("country")`

Descendants axis

```
PO...item
```

➔ `PO.Descendants("item")`

Late binding over XML

Child axis

```
BillTo.street
```

➔ `BillTo.Elements("street")`

Attribute axis

```
BillTo.@country
```

➔ `BillTo.Attribute("country")`

Descendants axis

```
PO...item
```

➔ `PO.Descendants("item")`

Static Duck Typing (typeclasses--)

```
Extension Customer For XElement
  Property Street As String
    Get
      Return CStr(Me.Element("street"))
    End Get
  End Property
  Property country As String
    Get
      Return CStr(Me.Attribute("street"))
    End Get
  End Property
  Property Street As IEnumerable(of Item)
    Get
      Return CType(Me.Descendants("item"), Item)
    End Get
  End Property
End Extension
```

Named extension

Compile-time cast

Conclusion

Programming Language Theory is relevant.
But it requires at least 20 years of patience and ripening.